Python OOP Exercises for Data Science & ML Engineering

Table of Contents

Basic OOP Concepts

Exercise 1: Basic Class and Objects

Objective: Understanding class definition, instance creation, and basic methods.

Create a `BankAccount` class with the following requirements:

- Attributes: account_number, holder_name, balance
- Methods: deposit(), withdraw(), check_balance(), get_account_info()
- Handle insufficient funds gracefully with custom exceptions
- Implement input validation for all operations

Expected Learning: Class definition, instance methods, basic encapsulation, exception handling

Exercise 2: Inheritance

Objective: Understanding inheritance and method overriding.

Create a `Vehicle` base class with common attributes (make, model, year, fuel_type). Create derived classes:

- `Car` with additional attributes (num_doors, transmission_type)
- `Motorcycle` with additional attributes (engine_size, bike_type)
- `ElectricVehicle` with attributes (battery_capacity, charging_time)

Implement method overriding for:

- `start_engine()` method (different for each vehicle type)
- `fuel_efficiency()` method
- `__str__()` method for proper string representation

Expected Learning: Inheritance, method overriding, polymorphism basics

Exercise 3: Encapsulation

Objective: Understanding private attributes and property decorators.

Design a `Student` class with:

- Private attributes: __student_id, __name, __grades (list)
- Public methods: add_grade(), get_average(), get_letter_grade()
- Property decorators for safe access to student information
- Validation: grades between 0-100, student_id format validation
- Method to calculate GPA on a 4.0 scale

Expected Learning: Private attributes, property decorators, data validation

Exercise 4: Polymorphism

Objective: Understanding polymorphism and abstract methods.

Create a `Shape` base class with abstract methods. Implement derived classes:

- `Circle` (radius)
- `Rectangle` (length, width)
- `Triangle` (base, height)
- `Square` (side) - inherits from Rectangle

Each class should implement:

- `area()` method
- `perimeter()` method
- `__str__()` method

Write a function that accepts a list of shapes and calculates total area and perimeter.

Expected Learning: Abstract methods, polymorphism, method implementation

Exercise 5: Class Methods and Static Methods

Objective: Understanding class vs instance vs static methods.

Create an `Employee` class with:

- Class variables: company_name, total_employees
- Instance variables: emp_id, name, salary, department
- Class methods: get_company_info(), get_employee_count()
- Static methods: validate_emp_id(), calculate_tax()
- Instance methods: give_raise(), transfer_department()

Expected Learning: Class methods, static methods, class variables

---

Advanced OOP Concepts

Exercise 6: Multiple Inheritance and Method Resolution Order (MRO)

Objective: Understanding complex inheritance hierarchies and MRO.

Create a diamond inheritance problem:

```
  A (Base)
 / \
B   C
 \ /
  D
```

Requirements:

- Class A: method `greet()` and `info()`
- Class B: inherits A, overrides `greet()`, adds `method_b()`
- Class C: inherits A, overrides `greet()`, adds `method_c()`
- Class D: inherits B and C, must resolve method conflicts

Tasks:

- Implement proper use of `super()` in all classes
- Print MRO for class D
- Handle method resolution conflicts gracefully
- Create a method that calls parent methods in specific order

Expected Learning: Multiple inheritance, MRO, super() usage, conflict resolution

Exercise 7: Metaclasses and Class Creation

Objective: Understanding metaclasses and dynamic class creation.

Create a custom metaclass `LoggingMeta` that:

- Automatically adds logging to all methods of a class
- Logs method entry with arguments
- Logs method exit with return values
- Logs execution time for each method
- Excludes magic methods from logging

Create a sample class using this metaclass and demonstrate its functionality.

Bonus: Create a metaclass that enforces naming conventions for class attributes.

Expected Learning: Metaclasses, dynamic method modification, decorators

Exercise 8: Descriptors and Property Decorators

Objective: Understanding descriptors and advanced property usage.

Create custom descriptors:

- `Temperature` descriptor: stores Celsius, accepts Fahrenheit input
- `ValidatedString` descriptor: validates string length and format
- `NumericRange` descriptor: validates numeric values within range

Use these descriptors in a `WeatherStation` class with attributes:

- temperature (Temperature descriptor)
- location (ValidatedString descriptor)
- humidity (NumericRange descriptor 0-100)
- pressure (NumericRange descriptor)

Also implement the same functionality using `@property` decorators and compare approaches.

Expected Learning: Descriptors, property decorators, data validation

Exercise 9: Context Managers and Resource Management

Objective: Understanding context managers and resource handling.

Create a custom context manager class `DatabaseConnection` that:

- Manages database connection pooling
- Implements automatic commit/rollback
- Handles connection timeouts
- Provides proper resource cleanup
- Supports nested transactions

Implement both approaches:

1. Using `__enter__` and `__exit__` methods
2. Using `@contextmanager` decorator

Expected Learning: Context managers, resource management, exception handling

Exercise 10: Abstract Base Classes and Protocol Design

Objective: Understanding ABC and interface design.

Design an abstract base class `PaymentProcessor` with:

- Abstract methods: `process_payment()`, `validate_payment()`, `refund_payment()`
- Concrete methods: `log_transaction()`, `send_receipt()`

Create concrete implementations:

- `CreditCardProcessor`
- `PayPalProcessor`
- `BankTransferProcessor`

Each should handle different validation rules and processing logic.

Expected Learning: Abstract base classes, interface design, polymorphism

Exercise 11: Decorator Pattern and Method Chaining

Objective: Understanding decorator pattern and fluent interfaces.

Create a `QueryBuilder` class that supports method chaining:

- `select(columns)` - specify columns
- `from_table(table)` - specify table
- `where(condition)` - add WHERE clause
- `order_by(column)` - add ORDER BY
- `limit(n)` - add LIMIT
- `execute()` - return final query string

Create decorators that can be applied to methods:

- `@cache_result` - caches method results
- `@log_execution` - logs method execution
- `@timing` - measures execution time

Expected Learning: Method chaining, decorator pattern, fluent interfaces

Exercise 12: Observer Pattern Implementation

Objective: Understanding the Observer pattern.

Implement the Observer pattern with:

- `Subject` class with methods: `attach()`, `detach()`, `notify()`
- `Observer` interface with `update()` method
- `NewsAgency` (subject) that notifies multiple `NewsChannel` observers
- Different types of observers: `EmailNotifier`, `SMSNotifier`, `PushNotifier`

Handle observer registration, removal, and notification with different message types.

Expected Learning: Observer pattern, event handling, loose coupling

Exercise 13: Custom Collections and Magic Methods

Objective: Understanding magic methods and custom collections.

Create a `Matrix` class that implements:

- Magic methods: `__add__`, `__mul__`, `__getitem__`, `__setitem__`, `__len__`
- Iterator protocol: `__iter__`, `__next__`
- String representation: `__str__`, `__repr__`
- Mathematical operations: transpose, determinant, inverse
- Comparison operators: `__eq__`, `__ne__`

Make it compatible with NumPy arrays and support broadcasting.

Expected Learning: Magic methods, iterator protocol, operator overloading

---

Data Science Specific OOP Exercises

Exercise 14: Data Pipeline Framework

Objective: Building reusable data processing pipelines.

Create a data pipeline framework with:

- `DataSource` abstract base class (CSV, JSON, Database sources)
- `DataProcessor` base class with common operations
- `DataValidator` class for data quality checks
- `DataTransformer` class for feature engineering
- `Pipeline` class that chains operations

Requirements:

- Support method chaining
- Handle different data formats
- Implement data validation at each step
- Support parallel processing
- Include logging and error handling

Example Usage:

python

```
pipeline = Pipeline()
    .add_source(CSVSource('data.csv'))
    .add_processor(DataCleaner())
    .add_transformer(FeatureEngineer())
    .add_validator(QualityChecker())
    .execute()
```

Exercise 15: Statistical Analysis Framework

Objective: Building a statistical analysis library.

Create a statistical analysis framework:

- `Dataset` class for data representation
- `StatisticalTest` abstract base class
- Concrete test classes: `TTest`, `ChiSquareTest`, `ANOVATest`
- `StatisticalReport` class for result presentation
- `Hypothesis` class for hypothesis testing

Features:

- Automatic test selection based on data types
- P-value calculation and interpretation
- Confidence intervals
- Effect size calculations
- Multiple comparison corrections

Exercise 16: Time Series Analysis Framework

Objective: Building time series analysis tools.

Create a time series framework:

- `TimeSeries` class with datetime indexing
- `TimeSeriesAnalyzer` with methods for:
  - Trend analysis
  - Seasonality detection
  - Outlier detection
  - Autocorrelation analysis
- `Forecaster` abstract base class
- Concrete forecasters: `ARIMAForecaster`, `ExponentialSmoothingForecaster`
- `TimeSeriesVisualizer` for plotting

Exercise 17: Feature Engineering Framework

Objective: Building automated feature engineering tools.

Create a feature engineering framework:

- `FeatureExtractor` abstract base class
- Concrete extractors: `NumericFeatures`, `CategoricalFeatures`, `TextFeatures`
- `FeatureSelector` class with various selection methods
- `FeatureTransformer` class for scaling and encoding
- `FeatureEngineeringPipeline` class

Features:

- Automatic feature type detection
- Missing value handling strategies
- Feature scaling and normalization
- Categorical encoding methods
- Text feature extraction (TF-IDF, n-grams)

Exercise 18: Experiment Tracking System

Objective: Building an ML experiment tracking system.

Create an experiment tracking system:

- `Experiment` class to represent ML experiments
- `ExperimentTracker` class to manage experiments
- `Metric` class for different evaluation metrics
- `Artifact` class for storing models and data
- `ExperimentComparator` for comparing experiments

Features:

- Parameter logging
- Metric tracking over time
- Model versioning
- Artifact storage
- Experiment comparison and visualization

ML Engineering OOP Exercises

Exercise 19: Model Registry System

Objective: Building a model management system.

Create a model registry system:

- `Model` class with metadata (version, performance, etc.)
- `ModelRegistry` class for model storage and retrieval
- `ModelValidator` class for model validation
- `ModelDeployer` class for model deployment
- `ModelMonitor` class for performance monitoring

Features:

- Model versioning
- A/B testing support
- Model performance tracking
- Automatic model validation
- Deployment pipeline integration

Exercise 20: Data Validation Framework

Objective: Building data quality assurance tools.

Create a data validation framework:

- `DataSchema` class for defining data expectations
- `Validator` abstract base class
- Concrete validators: `RangeValidator`, `TypeValidator`, `FormatValidator`
- `ValidationReport` class for reporting issues
- `DataQualityMonitor` class for continuous monitoring

Features:

- Schema definition and validation
- Data drift detection
- Anomaly detection
- Automated reporting
- Alert system for data quality issues

Exercise 21: Model Monitoring System

Objective: Building production model monitoring.

Create a model monitoring system:

- `ModelMonitor` class for tracking model performance
- `DriftDetector` class for detecting data/concept drift
- `PerformanceTracker` class for tracking metrics
- `AlertSystem` class for notifications
- `MonitoringDashboard` class for visualization

Features:

- Real-time performance monitoring
- Data drift detection
- Model degradation alerts
- Performance trend analysis
- Automated retraining triggers

Exercise 22: Feature Store Implementation

Objective: Building a feature store system.

Create a feature store:

- `Feature` class representing individual features
- `FeatureGroup` class for related features
- `FeatureStore` class for storage and retrieval
- `FeatureTransformation` class for real-time transformations
- `FeatureLineage` class for tracking feature dependencies

Features:

- Feature versioning
- Real-time and batch feature serving
- Feature lineage tracking
- Feature sharing across teams
- Feature quality monitoring

Bonus Challenges

Challenge 1: Design Pattern Integration

Objective: Combining multiple design patterns in a complex system.

Create a mini-framework that combines multiple design patterns in a `GameEngine`:

- **Factory Pattern**: For creating different game objects (enemies, weapons, items)
- **Strategy Pattern**: For different AI behaviors (aggressive, defensive, neutral)
- **Observer Pattern**: For event handling (player actions, game state changes)
- **Singleton Pattern**: For game state management
- **Command Pattern**: For user actions and undo functionality
- **State Pattern**: For game states (menu, playing, paused, game over)

Challenge 2: Distributed Computing Framework

Objective: Building a simplified distributed computing framework.

Create a distributed computing framework:

- `Task` class for representing computational tasks
- `Worker` class for processing tasks
- `Scheduler` class for task distribution
- `ResultCollector` class for aggregating results
- `ClusterManager` class for managing worker nodes

Features:

- Task serialization and distribution
- Fault tolerance and error handling
- Load balancing
- Result aggregation
- Monitoring and logging

Challenge 3: Real-time Data Processing Pipeline

Objective: Building a stream processing system.

Create a real-time data processing pipeline:

- `DataStream` class for representing data streams
- `StreamProcessor` abstract base class
- `WindowOperator` class for windowing operations
- `AggregationOperator` class for aggregations
- `FilterOperator` class for filtering
- `StreamingPipeline` class for chaining operations

Features:

- Real-time data ingestion
- Windowing operations (tumbling, sliding, session)
- Stream aggregations
- Event-time processing
- Fault tolerance and checkpointing

Learning Path Recommendations

For Beginners:

1. Start with Basic OOP Concepts (Exercises 1-5)
2. Move to simpler Advanced concepts (Exercises 6, 8, 10)
3. Try Data Science specific exercises (Exercises 14, 15)

For Intermediate:

1. Focus on Advanced OOP Concepts (Exercises 6-13)
2. Tackle Data Science specific exercises (Exercises 14-18)
3. Try some ML Engineering exercises (Exercises 19-20)

For Advanced:

1. Complete all Advanced OOP Concepts
2. Focus on ML Engineering exercises (Exercises 19-22)
3. Take on Bonus Challenges

Tips for Success:

- Implement each exercise step by step
- Test your implementations thoroughly
- Focus on clean, readable code
- Use type hints and docstrings
- Write unit tests for your classes
- Consider edge cases and error handling
- Practice code review and refactoring

Additional Resources

Recommended Reading:

- "Design Patterns: Elements of Reusable Object-Oriented Software" by Gang of Four
- "Clean Code" by Robert C. Martin
- "Effective Python" by Brett Slatkin
- "Python Tricks" by Dan Bader

Online Resources:

- Python Official Documentation
- Real Python tutorials
- Python Design Patterns documentation
- Data Science best practices guides

Practice Platforms:

- LeetCode (for algorithmic thinking)
- HackerRank (for Python-specific challenges)
- Kaggle (for data science applications)
- GitHub (for open source contribution)

This comprehensive exercise collection is designed to build strong OOP foundations while focusing on practical applications in data science and ML engineering. Work through the exercises systematically, and don't hesitate to experiment with variations and extensions.